



1. Introduction

1.1 Forward into the Past! (or a brief history of Struts)

When Java servlets were first invented, many programmers quickly realized that they were a Good Thing. They were faster and more powerful than standard CGI, portable, and infinitely extensible.

But writing HTML to send to the browser in endless `println()` statements was tiresome and problematic. The answer to that was [JavaServer Pages](#), which turned [Servlet](#) writing inside-out. Now developers could easily mix HTML with Java code, and have all the advantages of servlets. The sky was the limit!

Java web applications quickly became "JSP-centric". This in-and-of itself was not a Bad Thing, but it did little to resolve flow control issues and other problems endemic to web applications.

Another model was clearly needed ...

Many clever developers realized that JavaServer Pages AND servlets could be used **together** to deploy web applications. The servlets could help with the control-flow, and the JSPs could focus on the nasty business of writing HTML. In due course, using JSPs and servlets together became known as [Model 2](#) (meaning, presumably, that using JSPs alone was Model 1).

Of course, there is nothing new under the Sun ... and many have been quick to point out that JSP's Model 2 follows the classic [Model-View-Controller](#) design pattern abstracted from the venerable [Smalltalk MVC framework](#). Java Web developers now tend to use the terms Model 2 and MVC interchangeably. In this guide, we use the MVC paradigm to describe the Struts architecture, which might be best termed a Model 2/MVC design.

The Struts project was launched in May 2000 by Craig R. McClanahan to provide a standard MVC framework to the Java community. In July 2001, Struts 1.0 was released, and IOHO, Java Model 2 development will never be quite the same.

1.2 The Model-View-Controller ('MVC') Design Pattern

In the MVC design pattern, application flow is mediated by a central Controller. The Controller delegates requests - in our case, HTTP requests - to an appropriate handler. The handlers are tied to a Model, and each handler acts as an adapter between the request and the Model. The Model represents, or encapsulates, an application's business logic or state. Control is usually then forwarded back through the Controller to the appropriate View. The forwarding can be determined by consulting a set of mappings, usually loaded from a database or configuration file. This provides a loose coupling between the View and Model, which can make applications significantly easier to create and maintain.

1.2.1 The Model: System State and Business Logic JavaBeans

The *Model* portion of an MVC-based system can be often be divided into two major subsystems -- the **internal state** of the system and the **actions** that can be taken to change that state.

In grammatical terms, we might think about state information as **nouns** (things) and actions as **verbs** (changes to the state of those things).

Many applications represent the internal state of the system as a set of one or more JavaBeans. The bean properties represent the details of the system's state. Depending on your application's complexity, these beans may be self contained (and know how to persist their own state), or they may be facades that know how to retrieve the system's state from another component. This component may be a database, a search engine, an Entity Enterprise JavaBean, a LDAP server, or something else entirely.

Large-scale applications will often represent the set of possible business operations as methods that can be called on the bean or beans maintaining the state information. For example, you might have a shopping cart bean, stored in session scope for each current user, with properties that represent the current set of items that the user has decided to purchase. This bean might also have a `checkout()` method that authorizes the user's credit card and sends the order to the warehouse to be picked and shipped. Other systems will represent the available operations separately, perhaps as Session Enterprise JavaBeans (Session EJBs).

In a smaller scale application, on the other hand, the available operations might be embedded within the `Action` classes that are part of the Struts control layer. This can be useful when the logic is very simple or where reuse of the business logic in other environments is not contemplated.

The Struts framework architecture is flexible enough to support most any approach to accessing the Model, but we **strongly** recommend that you separate the business logic ("how it's done") from the role that `Action` classes play ("what to do"). 'nuff said.

For more about adapting your application's Model to Struts, see the [Building Model Components](#) chapter.

1.2.2 The View: JSP Pages and Presentation Components

The *View* portion of a Struts-based application is most often constructed using JavaServer Pages (JSP) technology. JSP pages can contain static HTML (or XML) text called "template text", plus the ability to insert dynamic content based on the interpretation (at page request time) of special action tags. The JSP environment includes a set of standard action tags, such as `<jsp:useBean>` whose purpose is described in the [JavaServer Pages Specification](#). In addition to the built-in actions, there is a standard facility to define your own tags, which are organized into "custom tag libraries."

Struts includes a set of custom tag libraries that facilitate creating user interfaces that are fully internationalized and interact gracefully with `ActionForm` beans. `ActionForms` capture and validate whatever input is required by the application.

For more about the Struts taglibs and using presentation pages with the framework, see the "[Building View Components](#)" chapter. Additional documentation regarding the taglibs is also available in the Developer Guides (see menu).

1.2.3 The Controller: `ActionServlet` and `ActionMapping`

The *Controller* portion of the application is focused on receiving requests from the client (typically a user running a web browser), deciding what business logic function is to be performed, and then delegating responsibility for producing the next phase of the user interface to an appropriate View component. In Struts, the primary component of the Controller is a servlet of class `ActionServlet`. This servlet is configured by defining a set of `ActionMappings`. An `ActionMapping` defines a path that is matched against the request URI of the incoming request and usually specifies the fully qualified class name of an Action class. All Actions are subclassed from `org.apache.struts.action.Action`. Actions encapsulate calls to business logic classes, interpret the outcome, and ultimately dispatch control to the appropriate View component to create the response.

Struts also supports the ability to use `ActionMapping` classes that have additional properties beyond the standard ones required to operate the framework. This allows you to store additional information specific to your application and still utilize the remaining features of the framework. In addition, Struts lets you define logical "names" to which control should be forwarded so that an action method can ask for the "Main Menu" page (for example), without knowing the location of the corresponding JSP page. These features greatly assist you in separating the control logic (what to do) with the view logic (how it's rendered).

For more about the Struts control layer, see the [Building Controller Components](#) chapter.

1.3 Struts Control Flow

The Struts framework provides several components that make up the **Control** layer of a MVC-style application. These include a controller servlet, developer-defined request handlers, and several supporting objects.

The Struts custom tag libraries provide direct support for the **View** layer of a MVC application. Some of these access the control-layer objects. Others are generic tags found convenient when writing applications. Other taglibs, including [JSTL](#), can also be used with Struts. Other presentation technologies, like [Velocity Templates](#) and [XSLT](#) can also be used with Struts.

The **Model** layer in a MVC application is often project-specific. Struts is designed to make it easy to access the business-end of your application, but leaves that part of the programming to other products, like [JDBC](#), [Enterprise Java Beans](#), [Object Relational Bridge](#), or [Simpler](#), to name a few.

Let's step through how this all fits together.

When initialized, the controller parses a configuration file (`struts-config.xml`) and uses it to deploy other control layer objects. Together, these objects form the **Struts Configuration**. The Struts Configuration defines (among other things) the [ActionMappings](#) [`org.apache.struts.action.ActionMappings`] for an application.

The Struts controller servlet consults the `ActionMappings` as it routes HTTP requests to other components in the framework. Requests may be forwarded to JavaServer Pages or [Action](#) [`org.apache.struts.action.Action`] subclasses provided by the Struts developer. Often, a request is first forwarded to an Action and then to a JSP (or other presentation page). The mappings help the controller turn HTTP requests into application actions.

An individual [ActionMapping](#) [`org.apache.struts.action.ActionMapping`] will usually contain a number of properties including:

- a **request path** (or "URI"),

- the **object type** (Action subclass) to act upon the request, and
- other properties as needed.

The Action object can handle the request and respond to the client (usually a Web browser) or indicate that control should be forwarded elsewhere. For example, if a login succeeds, a login action may wish to forward the request onto the mainMenu page.

Action objects have access to the application's controller servlet, and so have access to that servlet's methods. When forwarding control, an Action object can indirectly forward one or more shared objects, including [JavaBeans](#), by placing them in one of the standard contexts shared by Java Servlets.

For example, an Action object can create a shopping cart bean, add an item to the cart, place the bean in the session context, and then forward control to another mapping. That mapping may use a JavaServer Page to display the contents of the user's cart. Since each client has their own session, they will each also have their own shopping cart.

In a Struts application, most of the business logic can be represented using JavaBeans. An Action can call the properties of a JavaBean without knowing how it actually works. This encapsulates the business logic, so that the Action can focus on error handling and where to forward control.

JavaBeans can also be used to manage input forms. A key problem in designing Web applications is retaining and validating what a user has entered between requests. With Struts, you can define your own set of input bean classes, by subclassing [ActionForm](#) [org.apache.struts.action.ActionForm]. The ActionForm class makes it easy to store **and validate** the data for your application's input forms. The ActionForm bean is automatically saved in one of the standard, shared context collections, so that it can be used by other objects, like an Action object or another JSP.

The form bean can be used by a JSP to collect data from the user ... by an Action object to validate the user-entered data ... and then by the JSP again to re-populate the form fields. In the case of validation errors, Struts has a shared mechanism for raising and displaying error messages.

Another element of the Struts Configuration are the [ActionFormBeans](#) [org.apache.struts.action.ActionFormBeans]. This is a collection of [descriptor objects](#) that are used to create instances of the ActionForm objects at runtime. When a mapping needs an ActionForm, the servlet looks up the form-bean descriptor by name and uses it to create an ActionForm instance of the specified type.

Here is the sequence of events that occur when a request calls for an mapping that uses an ActionForm:

- The controller servlet either retrieves or creates the ActionForm bean instance.
- The controller servlet passes the bean to the Action object.
- If the request is being used to submit an input page, the Action object can examine the data. If necessary, the data can be sent back to the input form along with a list of messages to display on the page. Otherwise the data can be passed along to the business tier.
- If the request is being used to create an input page, the Action object can populate the bean with any data that the input page might need.

The Struts framework includes custom tags that can automatically populate fields from a JavaBean. All most JavaServer Pages really need to know about the rest of the framework is

the field names to use and where to submit the form.

Other Struts tags can automatically output messages queued by an Action or ActionForm and simply need to be integrated into the page's markup. The messages are designed for [localization](#) and will render the best available message for a user's locale.

The Struts framework and its custom tag libraries were designed from the ground-up to support the internationalization features built into the Java platform. All the field labels and messages can be retrieved from a [message resource](#). To provide messages for another language, simply add another file to the resource bundle.

Internationalism aside, other benefits to the message resources approach are consistent labeling between forms, and the ability to review all labels and messages from a central location.

For the simplest applications, an Action object may sometimes handle the business logic associated with a request. **However, in most cases, an Action object should invoke another object, usually a JavaBean, to perform the actual business logic.** This lets the Action focus on error handling and control flow, rather than business logic. To allow reuse on other platforms, business-logic JavaBeans should not refer to any Web application objects. The Action object should translate needed details from the HTTP request and pass those along to the business-logic beans as regular Java variables.

In a database application, for example:

- A business-logic bean will connect to and query the database,
- The business-logic bean returns the result to the Action,
- The Action stores the result in a form bean in the request,
- The JavaServer Page displays the result in a HTML form.

Neither the Action nor the JSP need to know (or care) from where the result comes. They just need to know how to package and display it.

[Other chapters](#) in this document cover the various Struts components in greater detail. The Struts release also includes several Developer Guides covering various aspects of the frameworks, along with sample applications, the standard Javadoc API, and, of course, the complete source code!

Struts is distributed under the [Apache Software Foundation license](#). The code is copyrighted, but is free to use in any application.

Next: [Building Model Components](#)



2. Building Model Components

2.1 Overview

Many requirements documents used for building web applications focus on the *View*. However, you should ensure that the processing required for each submitted request is also clearly defined from the *Model's* perspective. In general, the developer of the *Model* components will be focusing on the creation of JavaBeans classes that support all of the functional requirements. The precise nature of the beans required by a particular application will vary widely depending on those requirements, but they can generally be classified into several categories discussed below. However, a brief review of the concept of "scope" as it relates to beans and JSP is useful first.

2.2 JavaBeans and Scope

Within a web-based application, JavaBeans can be stored in (and accessed from) a number of different collections of "attributes". Each collection has different rules for the lifetime of that collection, and the visibility of the beans stored there. Together, the rules defining lifetime and visibility are called the *scope* of those beans. The JavaServer Pages (JSP) Specification defines scope choices using the following terms (with the equivalent servlet API concept defined in parentheses):

- **page** - Beans that are visible within a single JSP page, for the lifetime of the current request. (Local variables of the service method)
- **request** - Beans that are visible within a single JSP page, as well as to any page or servlet that is included in this page, or forwarded to by this page. (Request attributes)
- **session** - Beans that are visible to all JSP pages and servlets that participate in a particular user session, across one or more requests. (Session attributes)
- **application** - Beans that are visible to all JSP pages and servlets that are part of a web application. (Servlet context attributes)

It is important to remember that JSP pages and servlets in the same web application share the same sets of bean collections. For example, a bean stored as a request attribute in a servlet like this:

```
MyCart mycart = new MyCart(...);  
request.setAttribute("cart", mycart);
```

is immediately visible to a JSP page which this servlet forwards to, using a standard action tag like this:

```
<jsp:useBean id="cart" scope="request"  
class="com.mycompany.MyApp.MyCart"/>
```

2.3 ActionForm Beans

Note: While ActionForm beans often have properties that correspond to properties in your Model beans, the form beans themselves should be considered a Controller component. As such, they are able to transfer data between the Model and View layers.

The Struts framework generally assumes that you have defined an ActionForm bean (that is, a Java class extending the ActionForm class) for the input forms in your application. ActionForm beans are sometimes just called "form beans". These may be finely-grained objects, so that there is one bean for each form, or coarsely-grained so that one bean serves several forms, or even an entire application.

If you declare such beans in your Struts configuration file (see "[Building the Controller Components](#)"), the Struts controller servlet will automatically perform the following services for you, before invoking the appropriate Action method:

- Check for an instance of a bean of the appropriate class, under the appropriate key, in the appropriate scope (request or session).
- If there is no such bean instance available, a new one is automatically created and added to the appropriate scope (request or session).
- For every request parameter whose name corresponds to the name of a property in the bean, the corresponding setter method will be called. This operates in a manner similar to the standard JSP action `<jsp:setProperty>` when you use the asterisk wildcard to select all properties.
- The updated ActionForm bean will be passed to the execute method of an Action class [`org.apache.struts.Action`], so that the values can be made available to your system state and business logic beans.

For more about coding Actions and ActionForm beans, see the "[Building Controller Components](#)" chapter.

You should note that a "form", in the sense discussed here, does not necessarily correspond to a single JSP page in the user interface. It is common in many applications to have a "form" (from the user's perspective) that extends over multiple pages. Think, for example, of the wizard style user interface that is commonly used when installing new applications. Struts encourages you to define a single ActionForm bean that contains properties for all of the fields, no matter which page the field is actually displayed on. Likewise, the various pages of the same form should all be submitted to the same Action Class. If you follow these suggestions, the page designers can rearrange the fields among the various pages, often without requiring changes to the processing logic.

Smaller applications may only need a single ActionForm to service all of its input forms. Others applications might use a single ActionForm for each major subsystem of the application. Some teams might prefer to have a separate ActionForm class for each distinct input form or workflow. How many or how few ActionForms to use is entirely up to you. The framework doesn't care.

2.4 System State Beans

The actual state of a system is normally represented as a set of one or more JavaBeans classes, whose properties define the current state. A shopping cart system, for example, will include a bean that represents the cart being maintained for each individual shopper, and will (among other things) include the set of items that the shopper has currently selected for purchase. Separately, the system will also include different beans for the user's profile information (including their credit card and ship-to addresses), as well as the catalog of available items and their current inventory levels.

For small scale systems, or for state information that need not be kept for a long period of time, a set of system state beans may contain all the knowledge that the system ever has of these particular details. Or, as is often the case, the system state beans will represent information that is stored permanently in some external database (such as a `CustomerBean` object that corresponds to a particular row in the `CUSTOMERS` table), and are created or removed from the server's memory as needed. Entity Enterprise JavaBeans are also used for this purpose in large scale applications.

2.5 Business Logic Beans

You should encapsulate the functional logic of your application as method calls on JavaBeans designed for this purpose. These methods may be part of the same classes used for the system state beans, or they may be in separate classes dedicated to performing the logic. In the latter case, you will usually need to pass the system state beans to be manipulated to these methods as arguments.

For maximum code re-use, business logic beans should be designed and implemented so that they do **not** know they are being executed in a web application environment. If you find yourself having to import a `javax.servlet.*` class in your bean, you are tying this business logic to the web application environment. Consider rearranging things so that your `Action` classes (part of the Controller role, as described below) translate all required information from the HTTP request being processed into property setter calls on your business logic beans, after which a call to an `execute` method can be made. Such a business logic class can be reused in environments other than the web application for which they were initially constructed.

Depending on the complexity and scope of your application, business logic beans might be ordinary JavaBeans that interact with system state beans passed as arguments, or ordinary JavaBeans that access a database using JDBC calls. For larger applications, these beans will often be stateful or stateless Enterprise JavaBeans (EJBs) instead.

For more about using a database with your application, see the [Accessing a Database HowTo](#).

Next: [Building View Components](#)



3. Building View Components

3.1 Overview

This chapter focuses on the task of building the *View* components for use with the Struts framework. Many applications rely on JavaServer Pages (JSP) technology to create the presentation layer. The Struts distribution includes a comprehensive JSP tag library that provides support for building internationalized applications, as well as for interacting with input forms. Several other topics related to the View components are briefly discussed.

3.2 Internationalized Messages

A few years ago, application developers could count on having to support only residents of their own country, who are used to only one (or sometimes two) languages, and one way to represent numeric quantities like dates, numbers, and monetary values. However, the explosion of application development based on web technologies, as well as the deployment of such applications on the Internet and other broadly accessible networks, have rendered national boundaries invisible in many cases. This has translated (if you will pardon the pun) into a need for applications to support *internationalization* (often called "i18n" because 18 is the number of letters in between the "i" and the "n") and *localization*.

Struts builds upon the standard classes available on the Java platform to build internationalized and localized applications. The key concepts to become familiar with are:

- **Locale** - The fundamental Java class that supports internationalization is `Locale`. Each `Locale` represents a particular choice of country and language (plus an optional language variant), and also a set of formatting assumptions for things like numbers and dates.
- **ResourceBundle** - The `java.util.ResourceBundle` class provides the fundamental tools for supporting messages in multiple languages. See the Javadocs for the `ResourceBundle` class, and the information on Internationalization in the documentation bundle for your JDK release, for more information.
- **PropertyResourceBundle** - One of the standard implementations of `ResourceBundle` allows you to define resources using the same "name=value" syntax used to initialize properties files. This is very convenient for preparing resource bundles with messages that are used in a web application, because these messages are generally text oriented.
- **MessageFormat** - The `java.text.MessageFormat` class allows you to replace portions of a message string (in this case, one retrieved from a resource bundle) with arguments specified at run time. This is useful in cases where you are creating a sentence, but the words would appear in a different order in different languages. The placeholder string `{0}` in the message is replaced by the first runtime argument, `{1}` is replaced by the second argument, and so on.
- **MessageResources** - The Struts class `org.apache.struts.util.MessageResources` lets you treat a set of resource bundles like a database, and allows you to request a particular message string for a particular `Locale` (normally one associated with the current user) instead of for the default `Locale` the server itself is running in.

Please note that the i18n support in a framework like Struts is limited to the **presentation** of internationalized text and images to the user. Support for Locale specific **input methods** (used with languages such as Japanese, Chinese, and Korean) is left up to the client device, which is usually a web browser.

For an internationalized application, follow the steps described in the Internationalization document in the JDK documentation bundle for your platform to create a properties file containing the messages for each language. An example will illustrate this further:

Assume that your source code is created in package `com.mycompany.mypackage`, so it is stored in a directory (relative to your source directory) named `com/mycompany/mypackage`. To create a resource bundle called `com.mycompany.mypackage.MyApplication`, you would create the following files in the `com/mycompany/mypackage` directory:

- **MyApplication.properties** - Contains the messages in the default language for your server. If your default language is English, you might have an entry like this:
`prompt.hello=Hello`
- **MyApplication_xx.properties** - Contains the same messages in the language whose ISO language code is "xx" (See the [ResourceBundle Javadoc](#) page for a link to the current list). For a French version of the message shown above, you would have this entry:
`prompt.hello=Bonjour` You can have resource bundle files for as many languages as you need.

When you configure the controller servlet in the web application deployment descriptor, one of the things you will need to define in an initialization parameter is the base name of the resource bundle for the application. In the case described above, it would be `com.mycompany.mypackage.MyApplication`.

```
<servlet>
<servlet-name>action</servlet-name>
<servlet-class>
org.apache.struts.action.ActionServlet
</servlet-class>
<init-param>
<param-name>application</param-name>
<param-value>
    com.mycompany.mypackage.MyResources
</param-value>
</init-param>
<!-- ... -->
</servlet>
```

The important thing is for the resource bundle to be found on the class path for your application. Another approach is to store the `MyResources.properties` file in your application's `classes` folder. You can then simply specify "myResources" as the application value. Just be careful it is not deleted if your build script deletes classes as part of a "clean" target.

If it does, here is an Ant task to run when compiling your application that copies the contents of a `src/conf` directory to the `classes` directory:

```
<!-- Copy any configuration files -->
<copy todir="classes">
<fileset dir="src/conf"/>
</copy>
```

3.3 Forms and FormBean Interactions

Note: While the examples given here use JSP and custom tags, the ActionForm beans and the other Struts controller components are View neutral. Struts can be used with Velocity Templates, XSL, and any other presentation technology that can be rendered via a Java servlet.

At one time or another, most web developers have built forms using the standard capabilities of HTML, such as the `<input>` tag. Users have come to expect interactive applications to have certain behaviors, and one of these expectations relates to error handling -- if the user makes an error, the application should allow them to fix just what needs to be changed -- without having to re-enter any of the rest of the information on the current page or form.

Fulfilling this expectation is tedious and cumbersome when coding with standard HTML and JSP pages. For example, an input element for a username field might look like this (in JSP):

```
<input type="text" name="username"
value="<%= loginBean.getUsername() %>"/>
```

which is difficult to type correctly, confuses HTML developers who are not knowledgeable about programming concepts, and can cause problems with HTML editors. Instead, Struts provides a comprehensive facility for building forms, based on the Custom Tag Library facility of JSP 1.1. The case above would be rendered like this using Struts:

```
<html:text property="username"/>;
```

with no need to explicitly refer to the JavaBean from which the initial value is retrieved. That is handled automatically by the JSP tag, using facilities provided by the framework.

HTML forms are sometimes used to upload other files. Most browsers support this through a `<input type="file">` element, that generates a file browse button, but it's up to the developer to handle the incoming files. Struts handles these "multipart" forms in a way identical to building normal forms.

For an example of using Struts to create a simple login form, see the "[Building an ActionForm Howto](#)".

3.3.1 Indexed & Mapped Properties

Property references in JSP pages using the Struts framework can reference Java Bean properties as described in the JavaBeans specification. Most of these references refer to "scalar" bean properties, referring to primitive or single Object properties. However, Struts, along with the Jakarta Commons Beanutils library, allow you to use property references which refer to individual items in an array, collection, or map, which are represented by bean methods using well-defined naming and signature schemes.

Documentation on the Beanutils package can be found at <http://jakarta.apache.org/commons/beanutils/api/index.html>. More information about using indexed and mapped properties in Struts can be found in the FAQ describing [Indexed Properties, Mapped Properties, and Indexed Tags](#).

3.3.2 Input Field Types Supported

Struts defines HTML tags for all of the following types of input fields, with hyperlinks to the corresponding reference information.

- [checkboxes](#)
- [hidden](#) fields
- [password](#) input fields
- [radio](#) buttons
- [reset](#) buttons
- [select](#) lists with embedded option or options items
- [option](#)
- [options](#)
- [submit](#) buttons
- [text](#) input fields
- [textareas](#)

In every case, a field tag must be nested within a `form` tag, so that the field knows what bean to use for initializing displayed values.

3.3.3 Other Useful Presentation Tags

There are several tags useful for creating presentations, consult the documentation on each specific tag library, along with the Tag Developers Guides, for more information:

- [logic] [iterate](#) repeats its tag body once for each element of a specified collection (which can be an Enumeration, a Hashtable, a Vector, or an array of objects).
- [logic] [present](#) depending on which attribute is specified, this tag checks the current request, and evaluates the nested body content of this tag only if the specified value is present. Only one of the attributes may be used in one occurrence of this tag, unless you use the property attribute, in which case the name attribute is also required. The attributes include cookie, header, name, parameter, property, role, scope, and user.
- [logic] [notPresent](#) the companion tag to present, notPresent provides the same functionality when the specified attribute is not present.
- [html] [link](#) generates a HTML `<a>` element as an anchor definition or a hyperlink to the specified URL, and automatically applies URL encoding to maintain session state in the absence of cookie support.
- [html] [img](#) generates a HTML `<img>` element with the ability to dynamically modify the URLs specified by the "src" and "lowsrc" attributes in the same manner that `<html:link>` can.
- [bean] [parameter](#) retrieves the value of the specified request parameter, and defines the result as a page scope attribute of type String or String[].

3.3.4 Automatic Form Validation

In addition to the form and bean interactions described above, Struts offers an additional facility to validate the input fields it has received. To utilize this feature, override the following method in your `ActionForm` class:

```
validate(ActionMapping mapping,  
HttpServletRequest request);
```

The `validate` method is called by the controller servlet after the bean properties have been populated, but before the corresponding action class's `execute` method is invoked. The `validate` method has the following options:

- Perform the appropriate validations and find no problems -- Return either `null` or a zero-length `ActionErrors` instance, and the controller servlet will proceed to call the `perform` method of the appropriate `Action` class.
- Perform the appropriate validations and find problems -- Return an `ActionErrors` instance containing `ActionError`'s, which are classes that contain the error message keys (into the application's `MessageResources` bundle) that should be displayed. The controller servlet will store this array as a request attribute suitable for use by the `<html:errors>` tag, and will forward control back to the input form (identified by the `input` property for this `ActionMapping`).

As mentioned earlier, this feature is entirely optional. The default implementation of the `validate` method returns `null`, and the controller servlet will assume that any required validation is done by the action class.

One common approach is to perform simple, *prima facie* validations using the `ActionForm` `validate` method, and then handle the "business logic" validation from the `Action`.

The Struts Validator, covered in the next section, may be used to easily validate `ActionForms`.

3.3.5 The Struts Validator

Configuring the Validator to perform form validation is easy.

1. The `ActionForm` bean must extend `ValidatorForm`
2. The form's JSP must include the [<html:javascript>](#) tag for client side validation.
3. You must define the validation rules in an xml file like this:

```
<form-validation>
<formset>
<form name="logonForm">
  <field property="username" depends="required">
    <msg name="required" key="error.username"/>
  </field>
</form>
</formset>
</form-validation>
```

The `msg` element points to the message resource key to use when generating the error message.

4. Lastly, you must enable the `ValidatorPlugin` in the `struts-config.xml` file like this:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property
property="pathnames"
value="/WEB-INF/validator-rules.xml,
/WEB-INF/validation.xml"/>
</plug-in>
```

Note: If your required form property is one of the Java object representations of primitive types (ie. `java.lang.Integer`), you must set the `ActionServlet`'s `convertNull` init. parameter to `true`. Failing to do this will result in the required validation not being performed on that field because it will default to 0.

For more about the Struts Validator, see the [Developers Guide](#).

3.4 Other Presentation Techniques

Although the look and feel of your application can be completely constructed based on the standard capabilities of JSP and the Struts custom tag library, you should consider employing other techniques that will improve component reuse, reduce maintenance efforts, and/or reduce errors. Several options are discussed in the following sections.

3.4.1 Application-Specific Custom Tags

Beyond using the custom tags provided by the Struts library, it is easy to create tags that are specific to the application you are building, to assist in creating the user interface. The MailReader example application included with Struts illustrates this principle by creating the following tags unique to the implementation of this application:

- **checkLogon** - Checks for the existence of a particular session object, and forwards control to the logon page if it is missing. This is used to catch cases where a user has bookmarked a page in the middle of your application and tries to bypass logging on, or if the user's session has been timed out. (Note that there are better ways to authenticate users; the checkLogon tag is simply meant to demonstrate writing your own custom tags.)
- **linkSubscription** - Generates a hyperlink to a details page for a Subscription, which passes the required primary key values as request attributes. This is used when listing the subscriptions associated with a user, and providing links to edit or delete them.
- **linkUser** - Generates a hyperlink to a details page for a User, which passes the required primary key values as request attributes.

The source code for these tags is in the `src/example` directory, in package `org.apache.struts.example`, along with the other Java classes that are used in this application.

3.4.2 Page Composition With Includes

Creating the entire presentation of a page in one JSP file (with custom tags and beans to access the required dynamic data) is a very common design approach, and was employed in the example application included with Struts. However, many applications require the display of multiple logically distinct portions of your application together on a single page.

For example, a portal application might have some or all of the following functional capabilities available on the portal's "home" page:

- Access to a search engine for this portal.
- One or more "news feed" displays, with the topics of interest customized from the user's registration profile.
- Access to discussion topics related to this portal.
- A "mail waiting" indicator if your portal provides free email accounts.

The development of the various segments of this site is easier if you can divide up the work, and assign different developers to the different segments. Then, you can use the *include* capability of JavaServer Pages technology to combine the results into a single result page, or use the include tag provided with Struts. There are three types of *include* available, depending on when you want the combination of output to occur:

- An `<%@ include file="xxxxx" %>` directive can include a file that contains Java code

or JSP tags. The code in the included file can even reference variables declared earlier in the outer jsp page. The code is inlined into the other JavaServer Page before it is compiled so it can definitely contain more than just HTML.

- The include *action* (`<jsp:include page="xxxxxx" flush="true" />`) is processed at request time, and is handled transparently by the server. Among other things, that means you can conditionally perform the include by nesting it within a tag like [equal](#) by using its parameter attribute.
- The [bean:include](#) tag takes either a an argument "forward" representing a logical name mapped to the jsp to include, or the "id" argument, which represents a page context String variable to print out to the jsp page.

3.4.3 Page Composition With Tiles

Tiles is a powerful templating library that allows you to construct views by combining various "tiles". Here's a quick setup guide:

1. Create a `/layout/layout.jsp` file that contains your app's common look and feel:

```
<html>
<body>
<tiles:insert attribute="body"/>
</body>
</html>
```

2. Create your `/index.jsp` homepage file:

```
<h1>This is my homepage</h1>
```

3. Create a `/WEB-INF/tiles-defs.xml` file that looks like this:

```
<tiles-definitions>
<definition
  name="layout"
  path="/layout/layout.jsp">
  <put name="body" value=""/>
</definition>
<definition name="homepage" extends="layout">
  <put
    name="body"
    value="/index.jsp"/>
</definition>
</tiles-definitions>
```

4. Setup the TilesPlugin in the `struts-config.xml` file:

```
<plug-in
  className="org.apache.struts.tiles.TilesPlugin">
  <set-property
    property="definitions-config"
    value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

5. Setup an action mapping in `struts-config.xml` to point to your homepage tile:

```
<action
  path="/index"
```

```
type="org.apache.struts.actions.ForwardAction"  
parameter="homepage"/>
```

The TilesPlugin configures a special RequestProcessor that determines if the requested view is a tile and processes it accordingly. Note that we made the homepage tile extend our root layout tile and changed the body attribute. Tiles inserts the file named in the body attribute into the main layout.

See the tiles-documentation webapp for in-depth examples.

3.4.4 Image Rendering Components

Some applications require dynamically generated images, like the price charts on a stock reporting site. Two different approaches are commonly used to meet these requirements:

- Render a hyperlink with a URL that executes a servlet request. The servlet will use a graphics library to render the graphical image, set the content type appropriately (such as to image/gif), and send back the bytes of that image to the browser, which will display them just as if it had received a static file.
- Render the HTML code necessary to download a Java applet that creates the required graph. You can configure the graph by setting appropriate initialization parameters for the applet in the rendered code, or you can have the applet make its own connection to the server to receive these parameters.

3.4.5 Rendering Text

Some applications require dynamically generated text or markup, such as XML. If a complete page is being rendered, and can be output using a PrintWriter, this is very easy to do from an Action:

```
response.setContentType("text/plain"); // or text/xml  
PrintWriter writer = response.getWriter();  
// use writer to render text  
return(null);
```

3.4.6 The Struts-EL Tag Library

The **Struts-EL** tag library is a contributed library in the Struts distribution. It represents an integration of the Struts tag library with the JavaServer Pages Standard Tag Library, or at least the "expression evaluation" engine that is used by the JSTL.

The base Struts tag library contains tags which rely on the evaluation of "rtexprvalue"s (runtime scriptlet expressions) to evaluate dynamic attribute values. For instance, to print a message from a properties file based on a resource key, you would use the bean:write tag, perhaps like this:

```
<bean:message key='<%= stringvar %>' />
```

This assumes that stringvar exists as a JSP scripting variable. If you're using the **Struts-EL** library, the reference looks very similar, but slightly different, like this:

```
<bean-el:message key="{stringvar}"/>
```

If you want to know how to properly use the **Struts-EL** tag library, there are two important things you need to know:

- The Struts tag library
- The JavaServer Pages Standard tag library

Once you understand how to use these two, consider Struts tag attribute values being evaluated the same way the JSTL tag attribute values are. Past that, there is very little else you need to know to effectively use the **Struts-EL** tag library.

Although the **Struts-EL** tag library is a direct "port" of the tags from the Struts tag library, not all of the tags in the Struts tag library were implemented in the **Struts-EL** tag library. This was the case if it was clear that the functionality of a particular Struts tag could be entirely fulfilled by a tag in the JSTL. It is assumed that developers will want to use the **Struts-EL** tag library along with the JSTL, so it is reasonable to assume that they will use tags from the JSTL if they fill their needs.

For more see, [Struts-El Extension](#) in the FAQ/HOWTO section.

Next: [Building Controller Components](#)



4. Building Controller Components

4.1 Overview

Now that we understand how to construct the Model and View components of your application, it is time to focus on the Controller components. Struts includes a servlet that implements the primary function of mapping a request URI to an Action class. Therefore, your primary responsibilities related to the Controller are:

- Write an `ActionForm` class to mediate between the Model and the View. (See also [Building an ActionForm](#)).
- Write an `Action` class for each logical request that may be received (extend `org.apache.struts.action.Action`).
- Configure a `ActionMapping` (in XML) for each logical request that may be submitted. The XML configuration file is usually named `struts-config.xml`.

To deploy your application, you will also need to:

- Update the web application deployment descriptor file (in XML) for your application to include the necessary Struts components.
- Add the appropriate Struts components to your application.

The latter two items are covered in the "[Configuring Applications](#)" chapter.

4.2 The ActionServlet

For those of you familiar with MVC architecture, the `ActionServlet` represents the C - the controller. The job of the controller is to:

- process user requests,
- determine what the user is trying to achieve according to the request,
- pull data from the model (if necessary) to be given to the appropriate view, and
- select the proper view to respond to the user.

The Struts controller delegates most of this grunt work to the [Request Processor](#) and `Action` classes.

In addition to being the front controller for your application, the `ActionServlet` instance also is responsible for initialization and clean-up of resources. When the controller initializes, it first loads the application config corresponding to the "config" init-param. It then goes through an enumeration of all init-param elements, looking for those elements whose name starts with `config/`. For each of these elements, Struts loads the configuration file specified by the value of that init-param, and assigns a "prefix" value to that module's `ModuleConfig` instance consisting of the piece of the init-param name following "config/". For example, the module prefix specified by the init-param `config/foo` would be "foo". This is important to know, since this is how the controller determines which module will be given control of processing the request. To access the module foo, you would use a URL like:

`http://localhost:8080/myApp/foo/someAction.do`

For each request made of the controller, the method `process(HttpServletRequest, HttpServletResponse)` will be called. This method simply determines which module should service the request and then invokes that module's `RequestProcessor`'s `process` method, passing the same request and response.

4.2.1 Request Processor

The `RequestProcessor` is where the majority of the core processing occurs for each request. Let's take a look at the helper functions the `process` method invokes in-turn:

<code>processPath</code>	Determine the path that invoked us. This will be used later to retrieve an <code>ActionMapping</code> .
<code>processLocale</code>	Select a locale for this request, if one hasn't already been selected, and place it in the request.
<code>processContent</code>	Set the default content type (with optional character encoding) for all responses if requested.
<code>processNoCache</code>	If appropriate, set the following response headers: "Pragma", "Cache-Control", and "Expires".
<code>processPreprocess</code>	This is one of the "hooks" the <code>RequestProcessor</code> makes available for subclasses to override. The default implementation simply returns <code>true</code> . If you subclass <code>RequestProcessor</code> and override <code>processPreprocess</code> you should either return <code>true</code> (indicating process should continue processing the request) or <code>false</code> (indicating you have handled the request and the process should return)
<code>processMapping</code>	Determine the <code>ActionMapping</code> associated with this path.
<code>processRoles</code>	If the mapping has a role associated with it, ensure the requesting user has the specified role. If they do not, raise an error and stop processing of the request.
<code>processActionForm</code>	Instantiate (if necessary) the <code>ActionForm</code> associated with this mapping (if any) and place it into the appropriate scope.
<code>processPopulate</code>	Populate the <code>ActionForm</code> associated with this request, if any.
<code>processValidate</code>	Perform validation (if requested) on the <code>ActionForm</code> associated with this request (if any).
<code>processForward</code>	If this mapping represents a forward, forward to the path specified by the mapping.
<code>processInclude</code>	If this mapping represents an include, include the result of invoking the path in this request.
<code>processActionCreate</code>	Instantiate an instance of the class specified by the current <code>ActionMapping</code> (if necessary).
<code>processActionPerform</code>	This is the point at which your action's <code>perform</code> or <code>execute</code> method will be called.
<code>processForwardConfig</code>	Finally, the <code>process</code> method of the <code>RequestProcessor</code> takes the <code>ActionForward</code> returned by your <code>Action</code> class, and uses to select the next resource (if any). Most often the <code>ActionForward</code> leads to the presentation page that renders the response.

4.3 ActionForm Classes

An ActionForm represents an HTML form that the user interacts with over one or more pages. You will provide properties to hold the state of the form with getters and setters to access them. ActionForms can be stored in either the session (default) or request scopes. If they're in the session it's important to implement the form's `reset` method to initialize the form before each use. Struts sets the ActionForm's properties from the request parameters and sends the validated form to the appropriate Action's `execute` method.

When you code your ActionForm beans, keep the following principles in mind:

- The ActionForm class itself requires no specific methods to be implemented. It is used to identify the role these particular beans play in the overall architecture. Typically, an ActionForm bean will have only property getter and property setter methods, with no business logic.
- The ActionForm object also offers a standard validation mechanism. If you override a "stub" method, and provide error messages in the standard application resource, Struts will automatically validate the input from the form (using your method). See "[Automatic Form Validation](#)" for details. Of course, you can also ignore the ActionForm validation and provide your own in the Action object.
- Define a property (with associated `getXxx` and `setXxx` methods) for each field that is present in the form. The field name and property name must match according to the usual JavaBeans conventions (see the Javadoc for the `java.beans.Introspector` class for a start on information about this). For example, an input field named `username` will cause the `setUsername` method to be called.
- Buttons and other controls on your form can also be defined as properties. This can help determine which button or control was selected when the form was submitted. Remember, the ActionForm is meant to represent your data-entry form, not just the data beans.
- Think of your ActionForm beans as a firewall between HTTP and the Action. Use the `validate` method to ensure all required properties are present, and that they contain reasonable values. An ActionForm that fails validation will not even be presented to the Action for handling.
- You may also place a bean instance on your form, and use nested property references. For example, you might have a "customer" bean on your ActionForm, and then refer to the property "customer.name" in your presentation page. This would correspond to the methods `customer.getName()` and `customer.setName(string Name)` on your customer bean. See the Tag Library Developer Guides for more about using nested syntax with the Struts JSP tags.
- *Caution:* If you nest an existing bean instance on your form, think about the properties it exposes. Any public property on an ActionForm that accepts a single String value can be set with a query string. It may be useful to place beans that can affect the business state inside a thin "wrapper" that exposes only the properties required. This wrapper can also provide a filter to be sure runtime properties are not set to inappropriate values.

4.3.1 DynaActionForm Classes

Maintaining a separate concrete ActionForm class for each form in your Struts application is time-consuming. It is particularly frustrating when all the ActionForm does is gather and validate simple properties that are passed along to a business JavaBean.

This bottleneck can be alleviated through the use of DynaActionForm classes. Instead of creating a new ActionForm subclass and new get/set methods for each of your bean's properties, you can list its properties, type, and defaults in the Struts configuration file.

For example, add the following to `struts-config.xml` for a `UserForm` bean that stores a user's given and family names:

```
<form-bean
  name="UserForm"
  type="org.apache.struts.action.DynaActionForm">
  <form-property
    name="givenName"
    type="java.lang.String"
    initial="John"/>
  <form-property
    name="familyName"
    type="java.lang.String"
    initial="Smith"/>
</form-bean>
```

The types supported by `DynaActionForm` include:

- `java.lang.BigDecimal`
- `java.lang.BigInteger`
- `boolean` and `java.lang.Boolean`
- `byte` and `java.lang.Byte`
- `char` and `java.lang.Character`
- `java.lang.Class`
- `double` and `java.lang.Double`
- `float` and `java.lang.Float`
- `int` and `java.lang.Integer`
- `long` and `java.lang.Long`
- `short` and `java.lang.Short`
- `java.lang.String`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

You may also specify Arrays of these types (e.g. `String[]`). You may also specify a concrete implementation of the `Map` Interface, such as `java.util.HashMap`, or a `List` implementation, such as `java.util.ArrayList`.

If you do not supply an initial attribute, numbers will be initialized to 0 and objects to `null`.

In JSP pages using the original Struts custom tags, attributes of `DynaActionForm` objects can be referenced just like ordinary `ActionForm` objects. Wherever a Struts tag refers to a "property", the tags will automatically use the `DynaActionForm` properties just like those of a conventional `JavaBean`. You can even expose `DynaActionForm` properties using `bean:define`. (Although, you can't use `bean:define` to **instantiate** a `DynaActionForm`, since it needs to be setup with the appropriate `dyna-properties`).

If you are using the Struts JSTL EL taglib, the references are different, however. Only properties of ordinary `ActionForm` objects can be directly accessed through the JSTL expression language syntax. The `DynaActionForm` properties must be accessed through a slightly different syntax. The JSTL EL syntax for referencing a property of an `ActionForm` goes like this:

```
${formbean.prop}
```

The syntax for referencing a property of a `DynaActionForm` would be:

```
${dynabean.map.prop}
```

The map property is a property of `DynaActionForm` which represents the `HashMap` containing the `DynaActionForm` properties.

`DynaActionForms` are meant as an easy solution to a common problem: *Your ActionForms use simple properties and standard validations, and you just pass these properties over to another JavaBean* (say using `BeanUtils.copyProperties(myBusinessBean, form)`).

`DynaActionForms` are **not** a drop-in replacement for `ActionForms`. If you need to access `ActionForm` properties in your `Action`, you will need to use the map-style accessor, like `myForm.get("name")`. If you actively use the `ActionForm` object in your `Action`, then you may want to use conventional `ActionForms` instead.

`DynaActionForms` cannot be instantiated using a no-argument constructor. In order to simulate the extra properties, there is a lot of machinery involved in their construction. You must rely on Struts to instantiate a `DynaActionForm` for you, via the `ActionMapping`.

If need be, you can extend the `DynaActionForm` to add custom validate and reset methods you might need. Simply specify your subclass in the struts-config instead. However, you cannot mix conventional properties and `DynaProperties`. A conventional getter or setter on a `DynaActionForm` won't be found by the reflection utilities.

To use `DynaActionForms` with the Struts Validator, specify `org.apache.struts.validator.ValidatorActionForm` (or your subclass) as the form-bean class.

And, of course, while the `DynaActionForm` may support various binary types, properties used with the `html:text` tag should still be `String` properties.

`DynaActionForms` relieve developers of maintaining simple `ActionForms`. For even less maintenance, try Niall Pemberton's [LazyActionForm](#).

4.3.2 Map-backed ActionForms

The `DynaActionForm` classes offer the ability to create `ActionForm` beans at initialization time, based on a list of properties enumerated in the Struts configuration file. However, many HTML forms are generated dynamically at request time. Since the properties of these forms' `ActionForm` beans are not all known ahead of time, we need a new approach.

Struts allows you to make one or more of your `ActionForm`'s properties' values a `Map` instead of a traditional atomic object. You can then store the data from your form's dynamic fields in that `Map`. Here is an example of a map-backed `ActionForm` class:

```
public FooForm extends ActionForm {
    private final Map values = new HashMap();

    public void setValue(String key, Object value) {
        values.put(key, value);
    }

    public Object getValue(String key) {
        return values.get(key);
    }
}
```

In its corresponding JSP page, you can access objects stored in the values map using a special notation: `mapname(keyname)`. The parentheses in the bean property name indicate that:

- The bean property named mapname is indexed using Strings (probably backed by a Map), and that
- Struts should look for get/set methods that take a String key parameter to find the correct sub-property value. Struts will, of course, use the keyname value from the parentheses when it calls the get/set methods.

Here is a simple example:

```
<html:text property="value(foo)"/>
```

This will call the `getValue` method on `FooForm` with a key value of "foo" to find the property value. To create a form with dynamic field names, you could do the following:

```
<%
    for (int i = 0; i < 10; i++) {
        String name = "value(foo-" + i + ")";
%>
        <html:text property="<%= name %>" />
        <br/>
<%
    }
%>
```

Note that there is nothing special about the name value. Your map-backed property could instead be named `property`, `thingy`, or any other bean property name you prefer. You can even have multiple map-backed properties on the same bean.

In addition to map-backed properties, you can also create list-backed properties. You do so by creating indexed get/set methods on your bean:

```
public FooForm extends ActionForm {
    private final List values = new ArrayList();

    public void setValue(int key, Object value) {
        values.set(key, value);
    }

    public Object getValue(int key) {
        return values.get(key);
    }
}
```

In your presentation pages, you access individual entries in a list-backed property by using a different special notation: `listname[index]`. The braces in the bean property name indicate that the bean property named `listname` is indexed (probably backed by a List), and that Struts should look for get/set methods that take an index parameter in order to find the correct sub-property value.

While map-backed ActionForms provide you with more flexibility, they do not support the same range of syntax available to conventional or DynaActionForms. You might have difficulty referencing [indexed or mapped properties](#) using a map-backed ActionForm. The `validwhen` validator (since Struts 1.2.1) also does not support map-backed ActionForms.

4.4 Action Classes

The Action class defines two methods that could be executed depending on your servlet environment:

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             ServletRequest request,
                             ServletResponse response)
    throws Exception;

public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
    throws Exception;
```

Since the majority of Struts projects are focused on building web applications, most projects will only use the "HttpServletRequest" version. A non-HTTP `execute()` method has been provided for applications that are not specifically geared towards the HTTP protocol.

The goal of an `Action` class is to process a request, via its `execute` method, and return an `ActionForward` object that identifies where control should be forwarded (e.g. a JSP, Tile definition, Velocity template, or another `Action`) to provide the appropriate response. In the *MVC/Model 2* design pattern, a typical `Action` class will often implement logic like the following in its `execute` method:

- Validate the current state of the user's session (for example, checking that the user has successfully logged on). If the `Action` class finds that no logon exists, the request can be forwarded to the presentation page that displays the username and password prompts for logging on. This could occur because a user tried to enter an application "in the middle" (say, from a bookmark), or because the session has timed out, and the servlet container created a new one.
- If validation is not complete, validate the form bean properties as needed. If a problem is found, store the appropriate error message keys as a request attribute, and forward control back to the input form so that the errors can be corrected.
- Perform the processing required to deal with this request (such as saving a row into a database). This *can* be done by logic code embedded within the `Action` class itself, **but** should generally be performed by calling an appropriate method of a business logic bean.
- Update the server-side objects that will be used to create the next page of the user interface (typically request scope or session scope beans, depending on how long you need to keep these items available).
- Return an appropriate `ActionForward` object that identifies the presentation page to be used to generate this response, based on the newly updated beans. Typically, you will acquire a reference to such an object by calling `findForward` on either the `ActionMapping` object you received (if you are using a logical name local to this mapping), or on the controller servlet itself (if you are using a logical name global to the application).

In Struts 1.0, `Actions` called a `perform` method instead of the now-preferred `execute` method. These methods use the same parameters and differ only in which exceptions they throw. The elder `perform` method throws `ServletException` and `IOException`. The new `execute` method simply throws `Exception`. The change was to facilitate the Declarative Exception handling feature introduced in Struts 1.1.

The `perform` method may still be used in Struts 1.1 but is deprecated. The Struts 1.1 method simply calls the new `execute` method and wraps any `Exception` thrown as a `ServletException`.

4.4.1 Action Class Design Guidelines

Remember the following design guidelines when coding `Action` classes:

- **Write code for a multi-threaded environment** - The controller servlet creates **only one**

instance of your Action class, and uses this one instance to service all requests. Thus, you need to write thread-safe Action classes. Follow the same guidelines you would use to write thread-safe Servlets. Here are two general guidelines that will help you write scalable, thread-safe Action classes:

- **Only Use Local Variables** - The most important principle that aids in thread-safe coding is to use only local variables, **not instance variables**, in your Action class. Local variables are created on a stack that is assigned (by your JVM) to each request thread, so there is no need to worry about sharing them. An Action can be factored into several local methods, so long as all variables needed are passed as method parameters. This assures thread safety, as the JVM handles such variables internally using the call stack which is associated with a single Thread.
- **Conserve Resources** - As a general rule, allocating scarce resources and keeping them across requests from the same user (in the user's session) can cause scalability problems. For example, if your application uses JDBC and you allocate a separate JDBC connection for every user, you are probably going to run in some scalability issues when your site suddenly shows up on Slashdot. You should strive to use pools and release resources (such as database connections) prior to forwarding control to the appropriate View component -- even if a bean method you have called throws an exception.
- **Don't throw it, catch it!** - Ever used a commercial website only to have a stack trace or exception thrown in your face after you've already typed in your credit card number and clicked the purchase button? Let's just say it doesn't inspire confidence. Now is your chance to deal with these application errors - in the Action class. If your application specific code throws exceptions you should catch these exceptions in your Action class, log them in your application's log (`servlet.log("Error message", exception)`) and return the appropriate ActionForward.

It is wise to avoid creating lengthy and complex Action classes. If you start to embed too much logic in the Action class itself, you will begin to find the Action class hard to understand, maintain, and impossible to reuse. Rather than creating overly complex Action classes, it is generally a good practice to move most of the persistence, and "business logic" to a separate application layer. When an Action class becomes lengthy and procedural, it may be a good time to refactor your application architecture and move some of this logic to another conceptual layer; otherwise, you may be left with an inflexible application which can only be accessed in a web-application environment. Struts should be viewed as simply the **foundation** for implementing MVC in your applications. Struts provides you with a useful control layer, but it is not a fully featured platform for building MVC applications, soup to nuts.

The MailReader example application included with Struts stretches this design principle somewhat, because the business logic itself is embedded in the Action classes. This should be considered something of a bug in the design of the example, rather than an intrinsic feature of the Struts architecture, or an approach to be emulated. In order to demonstrate, in simple terms, the different ways Struts can be used, the MailReader application does not always follow best practices.

4.5 Exception Handler

You can define an `ExceptionHandler` to execute when an Action's `execute` method throws an `Exception`. First, you need to subclass `org.apache.struts.action.ExceptionHandler` and override the `execute` method. Your `execute` method should process the `Exception` and return an `ActionForward` object to tell Struts where to forward to next. Then you configure your handler in `struts-config.xml` like this:

```
<global-exceptions>
  <exception
```

```

    key="some.key"
    type="java.io.IOException"
    handler="com.yourcorp.ExceptionHandler"/>
</global-exceptions>

```

This configuration element says that `com.yourcorp.ExceptionHandler.execute` will be called when any `IOException` is thrown by an Action. The key is a key into your message resources properties file that can be used to retrieve an error message.

If the `handler` attribute is not specified, the default handler stores the exception in the request attribute under the value of the `Globals.EXCEPTION_KEY` global key.

The possible attributes for the "exception" element are as follows:

Attribute	Description	Default
<code>bundle</code>	Servlet context attribute for the message resources bundle associated with this handler. The default attribute is the value specified by the string constant declared at <code>Globals.MESSAGES_KEY</code> .	<code>org.apache.struts.Globals.MESSAGES_KEY</code>
<code>className</code>	The configuration bean for this <code>ExceptionHandler</code> object. If specified, <code>className</code> must be a subclass of the default configuration bean	<code>"org.apache.struts.config.ExceptionConfig"</code>
<code>extends</code>	The name of the exception handler that this will inherit configuration information from.	None
<code>handler</code>	Fully qualified Java class name for this exception handler.	<code>"org.apache.struts.action.ExceptionHandler"</code>
<code>key</code>	The key to use with this handler's message resource bundle that will retrieve the error message template for this exception.	None
<code>path</code>	The module-relative URI to the resource that will complete the request/response if this exception occurs.	None
<code>scope</code>	The context ("request" or "session") that is used to access the <code>ActionError</code> object [<code>org.apache.struts.action.ActionError</code>] for this exception.	<code>"request"</code>
<code>type</code>	Fully qualified Java class name of the exception type to register with this handler.	None

You can override global exception handlers by defining a handler inside an action definition.

A common use of `ExceptionHandler`s is to configure one for `java.lang.Exception` so it's called for any exception and log the exception to some data store.

4.6 Plugin Classes

The *PlugIn* interface extends *Action* and so that applications can easily hook into the *ActionServlet* lifecycle. This interface defines two methods, *init()* and *destroy()*, which are called at application startup and shutdown, respectively. A common use of a *Plugin Action* is to configure or load application-specific data as the web application is starting up.

At runtime, any resource setup by *init* would be accessed by *Actions* or business tier classes. The *PlugIn* interface allows you to setup resources, but does not provide any special way to access them. Most often, the resource would be stored in application context, under a known key, where other components can find it.

PlugIns are configured using `<plug-in>` elements within the Struts configuration file. See [PlugIn Configuration](#) for details.

4.7 The ActionMapping Implementation

In order to operate successfully, the Struts controller servlet needs to know several things about how each request URI should be mapped to an appropriate *Action* class. The required knowledge has been encapsulated in a Java class named *ActionMapping*, the most important properties are as follows:

- *type* - Fully qualified Java class name of the *Action* implementation class used by this mapping.
- *name* - The name of the form bean defined in the config file that this action will use.
- *path* - The request URI path that is matched to select this mapping. See below for examples of how matching works and how to use wildcards to match multiple request URIs.
- *unknown* - Set to *true* if this action should be configured as the default for this application, to handle all requests not handled by another action. Only one action can be defined as a default within a single application.
- *validate* - Set to *true* if the *validate* method of the action associated with this mapping should be called.
- *forward* - The request URI path to which control is passed when this mapping is invoked. This is an alternative to declaring a *type* property.

4.8 Writing Action Mappings

How does the controller servlet learn about the mappings you want? It would be possible (but tedious) to write a small Java class that simply instantiated new *ActionMapping* instances, and called all of the appropriate setter methods. To make this process easier, Struts uses the Jakarta Commons Digester component to parse an XML-based description of the desired mappings and create the appropriate objects initialized to the appropriate default values. See the [Jakarta Commons website](#) for more information about the Digester.

The developer's responsibility is to create an XML file named `struts-config.xml` and place it in the WEB-INF directory of your application. This format of this document is described by the Document Type Definition (DTD) maintained at http://struts.apache.org/dtds/struts-config_1_2.dtd. This chapter covers the configuration elements that you will typically write as part of developing your application. There are several other elements that can be placed in the `struts-config` file to customize your application. See "[Configuring Applications](#)" for more about the other elements in the Struts configuration file.

The controller uses an internal copy of this document to parse the configuration; an Internet

connection is not required for operation.

The outermost XML element must be `<struts-config>`. Inside of the `<struts-config>` element, there are three important elements that are used to describe your actions:

- `<form-beans>`
- `<global-forwards>`
- `<action-mappings>`

`<form-beans>`

This section contains your form bean definitions. Form beans are descriptors that are used to create `ActionForm` instances at runtime. You use a `<form-bean>` element for each form bean, which has the following important attributes:

- **name:** A unique identifier for this bean, which will be used to reference it in corresponding action mappings. Usually, this is also the name of the request or session attribute under which this form bean will be stored.
- **type:** The fully-qualified Java classname of the `ActionForm` subclass to use with this form bean.

`<global-forwards>`

This section contains your global forward definitions. Forwards are instances of the `ActionForward` class returned from an `ActionForm`'s `execute` method. These map logical names to specific resources (typically JSPs), allowing you to change the resource without changing references to it throughout your application. You use a `<forward>` element for each forward definition, which has the following important attributes:

- **name:** The logical name for this forward. This is used in your `ActionForm`'s `execute` method to forward to the next appropriate resource. Example: `homepage`
- **path:** The context relative path to the resource. Example: `/index.jsp` or `/index.do`
- **redirect:** True or false (default). Should the `ActionServlet` redirect to the resource instead of forward?

`<action-mappings>`

This section contains your action definitions. You use an `<action>` element for each of the mappings you would like to define. Most action elements will define at least the following attributes:

- **path:** The application context-relative path to the action.
- **type:** The fully qualified java classname of your `Action` class.
- **name:** The name of your `<form-bean>` element to use with this action

Other often-used attributes include:

- **parameter:** A general-purpose attribute often used by "standard" Actions to pass a required property.
- **roles:** A comma-delimited list of the user security roles that can access this mapping.

For a complete description of the elements that can be used with the `action` element, see the [Struts Configuration DTD](#) and the [ActionMapping documentation](#).

4.8.1 ActionMapping Example

Here's a mapping entry based on the MailReader example application. The MailReader application now uses `DynaActionForms`. But in this example, we'll show a conventional `ActionForm` instead, to illustrate the usual workflow. Note that the entries for all the other actions

are left out:

```
<struts-config>
  <form-beans>
    <form-bean
      name="logonForm"
      type="org.apache.struts.webapp.example.LogonForm" />
    </form-beans>
  <global-forwards>
    type="org.apache.struts.action.ActionForward">
      <forward
        name="logon"
        path="/logon.jsp"
        redirect="false" />
      </global-forwards>
    <action-mappings>
      <action
        path="/logon"
        type="org.apache.struts.webapp.example.LogonAction"
        name="logonForm"
        scope="request"
        input="/logon.jsp"
        unknown="false"
        validate="true" />
      </action-mappings>
    </struts-config>
```

First the form bean is defined. A basic bean of class "org.apache.struts.webapp.example.LogonForm" is mapped to the logical name "logonForm". This name is used as a request attribute name for the form bean.

The "global-forwards" section is used to create logical name mappings for commonly used presentation pages. Each of these forwards is available through a call to your action mapping instance, i.e. `mapping.findForward("logicalName")`.

As you can see, this mapping matches the path /logon (actually, because the MailReader example application uses extension mapping, the request URI you specify in a JSP page would end in /logon.do). When a request that matches this path is received, an instance of the *LogonAction* class will be created (the first time only) and used. The controller servlet will look for a bean in request scope under key logonForm, creating and saving a bean of the specified class if needed.

Optional but very useful are the local "forward" elements. In the MailReader example application, many actions include a local "success" and/or "failure" forward as part of an action mapping.

```
<!-- Edit mail subscription -->
<action
  path="/editSubscription"
  type="org.apache.struts.webapp.example.EditSubscriptionAction"
  name="subscriptionForm"
  scope="request"
  validate="false">
  <forward
    name="failure"
    path="/mainMenu.jsp"/>
  <forward
    name="success"
    path="/subscription.jsp"/>
</action>
```

Using just these two extra properties, the Action classes are almost totally independent of the actual names of the presentation pages. The pages can be renamed (for example) during a redesign, with negligible impact on the Action classes themselves. If the names of the "next"

pages were hard coded into the Action classes, all of these classes would also need to be modified. Of course, you can define whatever local forward properties makes sense for your own application.

The Struts configuration file includes several other elements that you can use to customize your application. See "[Configuring Applications](#)" for details.

4.9 Using ActionMappings for Pages

Fronting your pages with ActionMappings is *essential* when using modules, since doing so is the only way you involve the controller in the request -- and you want to! The controller puts the application configuration in the request, which makes available all of your module-specific configuration data (including which message resources you are using, request-processor, and so forth).

The simplest way to do this is to use the forward property of the ActionMapping:

```
<action path="/view" forward="/view.jsp"/>
```

4.10 Using Wildcards in ActionMappings

[Since Struts 1.2.0] As a Struts application grows in size, so will the number of action mappings. Wildcards can be used to combine similiar mappings into one more generic mapping.

The best way to explain wildcards is to show an example and walk through how it works. This example modifies the previous mapping in the [ActionMapping Example](#) section to use wildcards to match all pages that start with /edit:

```
<!-- Generic edit* mapping -->
<action
  path="/edit*"
  type="org.apache.struts.webapp.example.Edit{1}Action"
  name="{1}Form"
  scope="request"
  validate="false">
  <forward
    name="failure"
    path="/mainMenu.jsp"/>
  <forward
    name="success"
    path="/{1}.jsp"/>
</action>
```

The "*" in the path attribute allows the mapping to match the request URIs /editSubscription, editRegistration, or any other URI that starts with /edit, however /editSubscription/add would not be matched. The part of the URI matched by the wildcard will then be substituted into various attributes of the action mapping and its action forwards replacing {1}. For the rest of the request, Struts will see the action mapping and its action forwards containing the new values.

Mappings are matched against the request in the order they appear in the Struts configuration file. If more than one pattern matches the last one wins, so less specific patterns must appear before more specific ones. However, if the request URL can be matched against a path without any wildcards in it, no wildcard matching is performed and order is not important.

Wildcard patterns can contain one or more of the following special tokens:

*	Matches zero or more characters excluding the slash (/) character.
---	--

<code>**</code>	Matches zero or more characters including the slash ('/') character.
<code>\character</code>	The backslash character is used as an escape sequence. Thus <code>*</code> matches the character asterisk ('*'), and <code>\\</code> matches the character backslash ('\).

In the action mapping and action forwards, the wildcard-matched values can be accessed with the token `{N}` where `N` is a number from 1 to 9 indicating which wildcard-matched value to substitute. The whole request URI can be accessed with the `{0}` token.

The action mapping attributes that will accept wildcard-matched strings are:

- type
- name
- roles
- parameter
- attribute
- forward
- include
- input

Also, the action mapping properties (set using the `<set-property key="foo" value="bar">` syntax) will accept wildcard-matched strings in their `value` attribute.

The action forward attributes that will accept wildcard-matched strings are:

- path

4.11 Commons Logging Interface

Struts doesn't configure logging itself -- it's all done by [commons-logging](#) under the covers. The default algorithm is a search:

- If Log4J is there, use it.
- If JDK 1.4 is there, use it.
- Otherwise, use SimpleLog.

The commons-logging interface is an *ultra-thin* bridge to many different logging implementations. The intent is to remove compile- and run-time dependencies on any single logging implementation. For more information about the currently-supported implementations, please refer to the [the description for the org.apache.commons.logging package](#).

Because Struts uses commons-logging and, therefore, includes the necessary JAR files for **you** to use commons-logging, you've probably had the occasional fleeting thought, "*Should I use commons-logging?*" The answer (surprise!) depends on the requirements for your particular project. If one of your requirements is the ability to easily change logging implementations with zero impact on your application, then commons-logging is a very good option.

"Great! What do I do to get started using commons-logging in my own code?"

Using commons-logging in your own code is very simple - all you need are two imports and a declaration for a logger. Let's take a look:

```
package com.foo;
// ...
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
...
public class Foo {
```

```
// ...
private static Log log = LogFactory.getLog(Foo.class);
// ...
public void setBar(Bar bar) {
    if (log.isTraceEnabled()) {
        log.trace("Setting bar to " + bar);
    }
    this.bar = bar;
}
// ...
}
```

The general idea is to instantiate a single logger per class and to use a name for the logger which reflects where it's being used. The example is constructed with the class itself. This gives the logger the name of `com.foo.Foo`. Doing things this way lets you easily see where the output is coming from, so you can quickly pin-point problem areas. In addition, you are able to enable/disable logging in a very fine-grained way.

For examples of using logging in Struts classes, see the Action classes in the Struts MailReader example application.

Next: [Configuring Applications](#)



5. Configuring Applications

5.1 Overview

Before you can build an application, you need to lay a solid foundation. There are several setup tasks you need to perform before deploying your Struts application. These include components in the Struts configuration file and in the Web Application Deployment Descriptor.

5.2 The Struts configuration file

The [Building Controller Components](#) chapter covered writing the `<form-bean>` and `<action-mapping>` portions of the Struts configuration file. These elements usually play an important role in the development of a Struts application. The other elements in Struts configuration file tend to be static: you set them once and leave them alone.

These "static" configuration elements are:

- `<controller>`
- `<message-resources>`
- `<plug-in>`

5.2.1 Controller Configuration

The `<controller>` element allows you to configure the `ActionServlet`. Many of the controller parameters were previously defined by servlet initialization parameters in your `web.xml` file but have been moved to this section of `struts-config.xml` in order to allow different modules in the same web application to be configured differently. For full details on available parameters see the [struts-config_1_2.dtd](#) or the list below.

- **bufferSize** - The size (in bytes) of the input buffer used when processing file uploads. [4096] (optional)
- **className** - Classname of configuration bean. [org.apache.struts.config.ControllerConfig] (optional)
- **contentType** - Default content type (and optional character encoding) to be set on each response. May be overridden by the Action, JSP, or other resource to which the request is forwarded. [text/html] (optional)
- **forwardPattern** - Replacement pattern defining how the "path" attribute of a `<forward>` element is mapped to a context-relative URL when it starts with a slash (and when the `contextRelative` property is false). This value may consist of any combination of the following:
 - `$M` - Replaced by the module prefix of this module.
 - `$P` - Replaced by the "path" attribute of the selected `<forward>` element.
 - `$$` - Causes a literal dollar sign to be rendered.
 - `$x` - (Where "x" is any character not defined above) Silently swallowed, reserved for future use.

If not specified, the default `forwardPattern` is consistent with the previous behavior of forwards. [`$M$P`] (optional)

- **inputForward** - Set to true if you want the input attribute of `<action>` elements to be the name of a local or global `ActionForward`, which will then be used to calculate the ultimate URL. Set to false to treat the input parameter of `<action>` elements as a module-relative path to the resource to be used as the input form. [`false`] (optional)
- **locale** - Set to true if you want a `Locale` object stored in the user's session if not already present. [`true`] (optional)
- **maxFileSize** - The maximum size (in bytes) of a file to be accepted as a file upload. Can be expressed as a number followed by a "K", "M", or "G", which are interpreted to mean kilobytes, megabytes, or gigabytes, respectively. [`250M`] (optional)
- **multipartClass** - The fully qualified Java class name of the multipart request handler class to be used with this module.
[`org.apache.struts.upload.CommonsMultipartRequestHandler`] (optional)
- **nocache** - Set to true if you want the controller to add HTTP headers for defeating caching to every response from this module. [`false`] (optional)
- **pagePattern** - Replacement pattern defining how the page attribute of custom tags using it is mapped to a context-relative URL of the corresponding resource. This value may consist of any combination of the following:
 - `$M` - Replaced by the module prefix of this module.
 - `$P` - Replaced by the "path" attribute of the selected `<forward>` element.
 - `$$` - Causes a literal dollar sign to be rendered.
 - `$x` - (Where "x" is any character not defined above) Silently swallowed, reserved for future use.

If not specified, the default `pagePattern` is consistent with the previous behavior of URL calculation. [`$M$P`] (optional)

- **processorClass** - The fully qualified Java class name of the `RequestProcessor` subclass to be used with this module. [`org.apache.struts.chain.ComposableRequestProcessor`] (optional)
- **tempDir** - Temporary working directory to use when processing file uploads. [{the directory provided by the servlet container}]

This example uses the default values for several controller parameters. If you only want default behavior you can omit the controller section altogether.

```
<controller
  processorClass="org.apache.struts.action.RequestProcessor"
  contentType="text/html"/>;
```

5.2.2 Message Resources Configuration

Struts has built in support for internationalization (I18N). You can define one or more `<message-resources>` elements for your webapp; modules can define their own resource bundles. Different bundles can be used simultaneously in your application, the 'key' attribute is used to specify the desired bundle.

- **className** - Classname of configuration bean.
[`org.apache.struts.config.MessageResourcesConfig`] (optional)
- **factory** - Classname of `MessageResourcesFactory`.
[`org.apache.struts.util.PropertyMessageResourcesFactory`] (optional)
- **key** - ServletContext attribute key to store this bundle.
[`org.apache.struts.action.MESSAGE`] (optional)
- **null** - Set to false to display missing resource keys in your application like

'*???keyname???*' instead of `null`. [`true`] (optional)

- **parameter** - Name of the resource bundle. (required)

Example configuration:

```
<message-resources
  parameter="MyWebAppResources"
  null="false" />
```

This would set up a message resource bundle provided in the file `MyWebAppResources.properties` under the default key. Missing resource keys would be displayed as '*???keyname???*'.

5.2.3 Plugin Configuration

Struts Plugins are configured using the `<plug-in>` element within the Struts configuration file. This element has only one valid attribute, '`className`', which is the fully qualified name of the Java class which implements the `org.apache.struts.action.Plugin` interface.

For Plugins that require configuration themselves, the nested `<set-property>` element is available.

This is an example using the Tiles plugin:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
  <set-property
    property="definitions-config"
    value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

5.3 Configuring your application for modules

Very little is required in order to start taking advantage of the Struts module feature. Just go through the following steps:

1. Prepare a configuration file for each module.
2. Inform the controller of your module.
3. Use Actions to refer to your pages.

5.3.1 Module Configuration Files

Back in Struts 1.0, a few "boot-strap" options were placed in the `web.xml` file, and the bulk of the configuration was done in a single `struts-config.xml` file. Obviously, this wasn't ideal for a team environment, since multiple users had to share the same configuration file.

Since Struts 1.1, you have two options: you can list [multiple struts-config files](#) as a comma-delimited list, or you can subdivide a larger application into modules.

With the advent of modules, a given module has its own configuration file. This means each team (each module would presumably be developed by a single team) has their own configuration file, and there should be a lot less contention when trying to modify it.

5.3.2 Informing the Controller

Since Struts 1.0, you listed your configuration file as an initialization parameter to the action servlet in `web.xml`. This is still done since Struts 1.1, but the parameter can be extended. In order to tell the Struts machinery about your different modules, you specify multiple 'config' initialization parameters, with a slight twist. You'll still use 'config' to tell the `ActionServlet` about your "default" module, however, for each additional module, you will list an initialization parameter named "`config/module`", where `/module` is the prefix for your module (this gets used when determining which URIs fall under a given module, so choose something meaningful!). For example:

```
...
<init-param>
  <param-name>config</param-name>
  <param-value>/WEB-INF/conf/struts-default.xml</param-value>
</init-param>
<init-param>
  <param-name>config/module1</param-name>
  <param-value>/WEB-INF/conf/struts-module1.xml</param-value>
</init-param>
...
```

Here we have two modules. One happens to be the "default" module, identified by the param-name of "config", and the other will be using the module prefix `/module1` based on the param-name it was given ("`config/module1`"). The controller is configured to find the respective configuration files under `/WEB-INF/conf/` (which is the recommended place to put all configuration files). Pretty simple!

(The `struts-default.xml` would be equivalent to what most folks call `struts-config.xml`. I just like the symmetry of having all my Struts module configuration files being named `struts-module.xml`)

If you'd like to vary where the pages for each module are stored, see the [forwardPattern](#) setting for the Controller.

5.3.3 Switching Modules

There are three approaches for switching from one module to another. You can use the built-in `org.apache.struts.actions.SwitchAction`, you can use a `<forward>` (global or local) and specify the `contextRelative` attribute with a value of `true`, or you can specify the "module" parameter as part of any of the Struts hyperlink tags (`Include`, `Img`, `Link`, `Rewrite`, or `Forward`).

You can use `org.apache.struts.actions.SwitchAction` like so:

```
...
<action-mappings>
  <action path="/toModule"
    type="org.apache.struts.actions.SwitchAction"/>
  ...
</action-mappings>
...
```

Now, to change to `ModuleB`, we would use a URI like this:

```
http://localhost:8080/toModule.do?prefix=/moduleB&page=/index.do
```

If you are using the "default" module as well as "named" modules (like "/moduleB"), you can switch back to the "default" module with a URI like this:

```
http://localhost:8080/toModule.do?prefix=&page=/index.do
```

Here's an example of a global forward:

```
<global-forwards>
  <forward      name="toModuleB"
               contextRelative="true"
               path="/moduleB/index.do"
               redirect="true"/>
  ...
</global-forwards>
```

You could do the same thing with a local forward declared in an ActionMapping:

```
<action-mappings>
<action ... >
  <forward      name="success"
               contextRelative="true"
               path="/moduleB/index.do"
               redirect="true"/>
</action>
...
</action-mappings>
```

Or, you can use `org.apache.struts.actions.SwitchAction`:

```
<action-mappings>
<action path="/toModule"
        type="org.apache.struts.actions.SwitchAction"/>
...
</action-mappings>
```

Now, to change to ModuleB, we would use a URI like this:

```
http://localhost:8080/toModule.do?prefix=/moduleB&page=/index.do
```

Using the module parameter with a hyperlink tag is even simpler:

```
<html:link module="/moduleB" path="/index.do"/>
```

That's all there is to it! Happy module-switching!

5.4 The Web Application Deployment Descriptor

The final step in setting up the application is to configure the application deployment descriptor (stored in file `WEB-INF/web.xml`) to include all the Struts components that are required. Using the deployment descriptor for the example application as a guide, we see that the following

entries need to be created or modified.

5.4.1 Configure the ActionServlet Instance

Add an entry defining the action servlet itself, along with the appropriate initialization parameters. Such an entry might look like this:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>
      /WEB-INF/struts-config.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The initialization parameters supported by the action servlet are described below. (You can also find these details in the [Javadocs](#) for the ActionServlet class.) Square brackets describe the default values that are assumed if you do not provide a value for that initialization parameter.

- **config** - Context-relative path to the XML resource containing the configuration information for the default module. This may also be a comma-delimited list of configuration files. Each file is loaded in turn, and its objects are appended to the internal data structure. [/WEB-INF/struts-config.xml].
WARNING - If you define an object of the same name in more than one configuration file, the last one loaded quietly wins.
- **config/\${module}** - Context-relative path to the XML resource containing the configuration information for the application module that will use the specified prefix (/ \${module}). This can be repeated as many times as required for multiple application modules. (Since Struts 1.1)
- **convertNull** - Force simulation of the Struts 1.0 behavior when populating forms. If set to "true", the numeric Java wrapper class types (like java.lang.Integer) will default to null (rather than 0). (Since Struts 1.1) [false]
- **rulesets** - Comma-delimited list of fully qualified classnames of additional org.apache.commons.digester.RuleSet instances that should be added to the Digester that will be processing struts-config.xml files. By default, only the RuleSet for the standard configuration elements is loaded. (Since Struts 1.1)
- **validating** - Should we use a validating XML parser to process the configuration file (strongly recommended)? [true]
- **configFactory** - The Java class name of the ModuleConfigFactory used to create the implementation of the ModuleConfig interface.
[org.apache.struts.config.impl.DefaultModuleConfigFactory]
- **chainConfig** - Comma-separated list of either context-relative or classloader path(s) to load commons-chain catalog definitions from. If none specified, the default Struts catalog that is provided with Struts will be used. (Since Struts 1.3)

WARNING - Struts will not operate correctly if you define more than one <servlet> element for a controller servlet, or a subclass of the standard controller servlet class. The controller servlet **MUST** be a web application wide singleton.

5.4.2 Configure the ActionServlet Mapping

Note: The material in this section is not specific to Struts. The configuration of servlet mappings is defined in the Java Servlet Specification. This section describes the most common means of configuring a Struts application.

There are two common approaches to defining the URLs that will be processed by the controller servlet -- prefix matching and extension matching. An appropriate mapping entry for each approach will be described below.

Prefix matching means that you want all URLs that start (after the context path part) with a particular value to be passed to this servlet. Such an entry might look like this:

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>/do/*</url-pattern>
</servlet-mapping>
```

which means that a request URI to match the /logon path described earlier might look like this:

```
http://www.mycompany.com/myapplication/do/logon
```

where /myapplication is the context path under which your application is deployed.

Extension mapping, on the other hand, matches request URIs to the action servlet based on the fact that the URI ends with a period followed by a defined set of characters. For example, the JSP processing servlet is mapped to the *.jsp pattern so that it is called to process every JSP page that is requested. To use the *.do extension (which implies "do something"), the mapping entry would look like this:

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

and a request URI to match the /logon path described earlier might look like this:

```
http://www.mycompany.com/myapplication/logon.do
```

WARNING - Struts will not operate correctly if you define more than one <servlet-mapping> element for the controller servlet.

WARNING - If you are using the new module support since Struts 1.1, you should be aware that **only** extension mapping is supported.

5.4.3 Configure the Struts Tag Libraries

Next, you must add an entry defining the Struts tag libraries.

The [struts-bean](#) taglib contains tags useful in accessing beans and their properties, as well as defining new beans (based on these accesses) that are accessible to the remainder of the page via scripting variables and page scope attributes. Convenient mechanisms to create new beans based on the value of request cookies, headers, and parameters are also provided.

The [struts-html](#) taglib contains tags used to create Struts input forms, as well as other tags generally useful in the creation of HTML-based user interfaces.

The [struts-logic](#) taglib contains tags that are useful in managing conditional generation of output text, looping over object collections for repetitive generation of output text, and application flow management.

The [struts-tiles](#) taglib contains tags used for combining various view components, called "tiles", into a final composite view.

The [struts-nested](#) taglib is an extension of other struts taglibs that allows the use of nested beans.

Below is how you would define all Struts taglibs for use within your application. In practice, you would only specify the taglibs that your application uses:

```
<taglib>
  <taglib-uri>
    http://struts.apache.org/tags-bean
  </taglib-uri>
  <taglib-location>
    /WEB-INF/struts-bean.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    http://struts.apache.org/tags-html
  </taglib-uri>
  <taglib-location>
    /WEB-INF/struts-html.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    http://struts.apache.org/tags-logic
  </taglib-uri>
  <taglib-location>
    /WEB-INF/struts-logic.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    http://struts.apache.org/tags-tiles
  </taglib-uri>
  <taglib-location>
    /WEB-INF/struts-tiles.tld
  </taglib-location>
</taglib>
<taglib>
  <taglib-uri>
    http://struts.apache.org/tags-nested
  </taglib-uri>
  <taglib-location>
    /WEB-INF/struts-nested.tld
  </taglib-location>
</taglib>
```

This tells the JSP system where to find the tag library descriptor for this library (in your application's WEB-INF directory, instead of out on the Internet somewhere).

5.4.3.1 Configure the Struts Tag Libraries (Servlet 2.3/2.4)

Servlet 2.3/2.4 users only: The Servlet 2.3 and 2.4 specifications simplify the deployment and

configuration of tag libraries. The instructions above will work on older containers as well as 2.3 and 2.4 containers (Struts only requires a servlet 2.2 container); however, if you're using a 2.3/2.4 container such as Tomcat 4.x/5.x, you can take advantage of a simplified deployment.

All that's required to install the Struts tag libraries is to copy `struts.jar` into your `/WEB-INF/lib` directory and reference the tags in your code like this:

```
<%@ taglib
    uri="http://struts.apache.org/tags-html"
    prefix="html" %>
```

Note that you **must use the full uri** defined in the various tlds (see the [example configuration](#) for reference) so that the container knows where to find the tag's class files. You don't have to alter your `web.xml` file or copy tlds into any application directories.

5.5 Add Struts Components To Your Application

To use Struts, you must copy the `.tld` files that you require into your `WEB-INF` directory, and copy `struts.jar` (and all of the `commons-*.jar` files) into your `WEB-INF/lib` directory.

Servlet 2.3/2.4 Users: See [section 4.5.3.1](#) for how to avoid copying the tlds into your application.

Sidebar: Sharing JAR Files Across Web Applications

Many servlet containers and application servers provide facilities for sharing JAR files across multiple web applications that depend on them. For example, Tomcat 4.1 allows you to put JAR files into the `$CATALINA_HOME/shared/lib` or `$CATALINA_HOME/common/lib` directories, and the classes in those JAR files will be available in all applications, without the need to place them in every web application's `/WEB-INF/lib` directory. Usually, the sharing is accomplished by creating a separate class loader that is the parent of the class loader (created by your container) for each individual web application.

If you have multiple Struts-based web applications, it is tempting to consider taking advantage of this container feature, and placing `struts.jar` and the various `commons-*.jar` files in the shared directory, rather than in each web application. However, there are several potential, and actual, problems with this approach:

- Classes loaded from the shared class loader cannot see classes in the web application's class loader, unless they are specifically programmed to use the Thread context class loader. For example, Struts dynamically loads your action and form bean classes, and normally would not be able to find those classes. Struts has been programmed to deal with this in *most* scenarios, but it has not been thoroughly audited to ensure that it works in *all* scenarios. The Commons libraries that Struts uses have **NOT** been audited to catch all possible scenarios where this might become a problem.
- When a class is loaded from a shared class loader, static variables used within that class become global as well. This can cause

inter-webapp conflicts when the underlying code assumes that the statics are global only within a particular web applicaiton (which would be true if the class was loaded from the webapp class loader). There are many cases where Struts, and the Commons libraries it relies on, use static variables to maintain information that is presumed to be visible only within a single web applicaiton. Sharing these JAR files can cause unwanted interactions, and probably cause incorrect behavior.

- When JAR files are shared like this, it is not possible to update the JAR file versions employed by a single web application without updating all of them. In addition, because updating a Struts version normally requires recompilation of the applications that use it, you will have to recompile all of your applications as well, instead of being able to manage them independently.

In spite of these difficulties, it is possible that sharing the Struts and Commons JAR files *might* appear to work for you. However, this is **NOT** a supported configuration.

If you file a bug report for `ClassNotFoundException` or `NoClassDefFoundError` exceptions, or similar situations where it appears that the wrong version of a class is being loaded, the bug report will **NOT** be processed unless the problem exists with the JAR files in their recommended location, in the `/WEB-INF/lib` subdirectory of your webapp.

5.6 Logging in Struts Based Applications

Since Struts 1.0, the logging functionality was fairly limited. You could set a debugging detail level with a servlet initialization parameter, and all log messages were written to wherever `ServletContext.log()` output is sent by your servlet container. With Struts 1.1, however, all logging messages written by Struts itself, as well as the commons libraries that it utilizes, flow through an abstract wrapper called [Commons Logging](#), which can be used as a wrapper around any logging implementation. The most common implementations used are simple logging to `System.err`, the [Apache Log4J](#) package, or the built-in logging capabilities of JDK 1.4 or later in the [java.util.logging](#) package.

This section does not attempt to fully explain how Commons Logging is configured and used. Instead, it focuses on pertinent details of using Commons Logging in a Struts based environment. For complete documentation on using Commons Logging, consult the documentation for the logging system you are using, plus the Commons Logging [Javadocs](#).

Commons Logging provides fine-grained control over the logging messages created by a `Log` instance. By convention, the `Log` instances for Struts (and the Commons packages in general) are named the fully qualified class name of the class whose messages are being logged. Therefore, log messages created by the `RequestProcessor` class are, naturally enough, directed to a logger named `org.apache.struts.action.RequestProcessor`.

The advantage of this approach is that you can configure the level of detail in the output you want from each class, individually. However, it would be a burden to be required to maintain such settings for every possible class, so the logging environment supports the notion of logging *hierarchies* as well. If a detail level configuration for a particular class has not been set, the logging system looks up the hierarchy until it finds a configuration setting to use, or else uses the default detail level if no configuration for any level of the hierarchy has been

explicitly set. In the case of our messages from `RequestProcessor`, the logging system will look for explicit settings of the following loggers, in this order, until it finds one:

- `org.apache.struts.action.RequestProcessor`
- `org.apache.struts.action`
- `org.apache.struts`
- `org.apache`
- `org`
- The default logging detail level for your log implementation.

In a similar manner, the detail level for messages from `PropertyUtils` (from the Commons BeanUtils library) is set by a search for configuration settings for:

- `org.apache.commons.beanutils.PropertyUtils`
- `org.apache.commons.beanutils`
- `org.apache.commons`
- `org.apache`
- `org`
- The default logging detail level for your log implementation.

You can seamlessly integrate logging from your own components into the same logging implementation that Struts and the Commons libraries use, by following the instructions in [Section 4.10](#). If you do this, you are strongly encouraged to follow the same naming convention for loggers (based on the class name of the messages being logged) for maximum configuration flexibility.

For more about putting it all together, see the [Building Applications HowTo](#).

Next: [Release Notes](#)